Under the hood of Cairo 1.0: Exploring Sierra

Part 3: Become a better Cairo developer with Sierra

Mathieu

Follow

Nethermind.eth

--

Listen

Share

Introduction

In the first blog post of the series, we introduced Sierra, an intermediate language designed to simplify the development process of Starknet contracts by ensuring that all deployed code on Starknet cannot produce errors at runtime. In the second post, we analyzed the structure of a Sierra program to provide a comprehensive understanding of Sierra and why it is a safe intermediate representation of the Cairo code. In our closing article, we'll delve deeper into some core and novel concepts introduced in Cairo 1 by analyzing Sierra code.

Mutable variables, References, and Snapshots

Cairo introduces a new idea of snapshots which is often confused with references. There are three different ways of passing variables in function calls in Cairo: pass-by-value, where the caller function takes ownership of the variable; pass-by-reference, where the caller function "borrows" the variable and returns ownership to the caller context after its execution; and pass-by-snapshot, where you create a snapshot of a value, which is an immutable view to a value, and pass it to the function so that you keep ownership of the base value.

The Rust equivalents of ref

and @

(snapshots) would be ref <=> &mut, @ <=> &

, but there are subtleties to be aware of. Special care must be put into understanding how mutable references can be passed as function parameters and how values are mutated. Snapshots are a concept exclusive to Cairo 1 and do not have a direct equivalent in other languages! Looking at Sierra code will give us a better understanding.

Mutable variables

Let's start with a simple concept - mutable variables. In a traditional programming language, each variable is associated with a specific memory cell, a location in the computer's memory where the variable's data is stored. When a variable is assigned a value, the value is stored in the memory cell associated with that variable. The variable can then access or modify the value stored in that memory cell throughout the program's execution.

However, in Cairo, it's impossible to modify the content of a memory cell that has already been written to. Analyzing the compiled Sierra code, let's see what exactly happens when you declare a variable as mut in a Cairo program. Let's consider the Cairo program, in which a variable x

is declared as mut

and y

is not, and we shadow the previous declaration of y

.

The Sierra code it compiles to is:

As demonstrated in this compiled Sierra program, mutable variables are syntactic sugar that enables Cairo developers to effortlessly modify and update data values throughout a program's execution without having to manually shadow the previously declared variable. When we modify our mutable variable x

, the corresponding Sierra variable storing its value is first dropped as it's no longer used, and then a new variable is created with the updated value, as shown on lines 13-14. Similarly, for our non-mutable variable y

, whose value is shadowed, the procedure in Sierra is exactly the same: the prior value is dropped, and a new one is instantiated with the updated value associated with y

, as shown on lines 17-18. It is, however, recommended to use mutable variables instead of shadowing where possible, as it ensures consistency in types.

The Unit

type declared represents an empty Struct and is the type returned by default by functions that don't return values.

References

In traditional languages, “pass-by-reference” is a method of passing variables to functions where the function receives a reference to the variable's memory location. This allows the function to modify the variable's value directly. In Cairo, the equivalent is achieved using the ref

modifier when defining the function parameter. However, as previously stated, it's essential to note that once assigned variable values can't be modified directly in Cairo, unlike in other languages.

Consider the following code snippet in Cairo:

In this example, the x

variable is defined as mutable using the mut

keyword, and a mutable reference to x

is passed to the increment

function using the ref

prefix. The function directly increments the value of x

, and the new value is returned.

To further understand how this operates at a lower level, let's analyze the corresponding Sierra code:

The first observation here is the signature of the increment

function from its declaration in Cairo. As expected, the function returns the default Unit

type for functions without return values, which is anticipated. However, it also returns a felt252

. When function parameters are declared as ref

, the compiler will generate code to automatically return the updated value of the argument passed to the function without the need to specify it in the higher-level code.

This is another example of how Cairo provides syntactic sugar to improve the developer experience. The above code is functionally equivalent to the following pass-by-value code, and they compile roughly the same Sierra code.

Snapshots

In the Cairo programming language, snapshots are introduced as a wrapper type that creates an immutable view of an object at a given time. Snapshots are useful when we need to perform on non-duplicable types like arrays. In runtime implementation, snapshots are zero-cost abstraction because of Cairo Assembly’s write-once memory model.

To understand more about what snapshots are and how to use them, let's see how they are compiled to Sierra. In the following Cairo program, we define a variable x

and pass a snapshot of x

to the pass_by_snapshot

function, which returns the value of x

using the desnap operator *

.

While analyzing this program, we observe:

- The snapshot type doesn't exist in this Sierra program. Instead, the Sierra code only uses felt252

. The pass_by_snapshot

signature takes a felt252

as a parameter, even though we specified in our Cairo program that it should take a snapshot as a parameter.

- The snapshot_take

libfunc takes a felt252

as input and returns two variables. Its signature is very similar to the dup

libfunc.

- The desnap operator *

doesn't generate any Sierra code

To understand more about what's happening here, let's dive into the compiler code. In the cairo-lang-sierra crate, we learn that a snapshot is just a wrapper around an object that ensures the original object is not modified. The snapshot_take

libfunc only returns a snapshot to the type if the type cannot be copied

. Duplicatable types are their own snapshot - as the snapshot itself is useless if we can duplicate the value. This concept of snapshots only exists at the Sierra level and makes the linear type system effective by ensuring that the object wrapped in a snapshot can't be modified.

But when do we find snapshots particularly useful? Specifically when working with non-duplicable types like Arrays. In the following code, a function foo

takes as a parameter an Array a

. A snapshot to this array is passed to two functions, and the array is then returned.

In the generated Sierra code, we note the declaration a Snapshot type. Unlike our previous example, the snapshot_take

libfunc returns both the original object and a snapshot of the original object - a wrapper type around our object. This snapshot is then passed to our functions. If you attempt to call a function that modifies the array object, such as the array_append

libfunc, the Sierra program will not compile to CASM because a type mismatch will be detected at compile time. This is because you are attempting to append to a Snapshot>

type, but the array_append

libfunc expects an Array type

.

In summary, when a function takes a snapshot to a value using @

, it is only able to read the value and not modify it. It behaves like an immutable borrow using &

in Rust, which allows multiple parts of the program to read the same value simultaneously while ensuring that it is not modified. When working with non-copyable objects, using snapshots allows you to retain ownership of the object in the calling context while ensuring the object remains unaltered.

Function inlining

Function inlining is a compiler optimization technique that substitutes a function call with the actual code of the function being called. It eliminates the overhead of a function call by integrating the function's code directly into the calling function.

The Cairo compiler will automatically replace calls to functions marked as inline directly with their Sierra code. This optimization is especially useful for frequently called small functions. Inlining can reduce the overhead of function calls and lead to faster and more optimised executions, as values don't need to be pushed to memory. Consider the Cairo program

where the first function has an [inline(always)]

attribute, while the second doesn't.

In the Sierra code resulting from this program, instead of executing the inline function using a function_call

libfunc to execute the inline

function at line 15, the compiler integrates the code directly into the main function.

However, using function inlining can increase the overall program size due to code duplication for each inlined function call. Therefore, it is recommended to use inlining only for frequently called functions that have a limited number of instructions.

Conclusion

In this post, we have explored some core concepts of Cairo 1, like mutable variables, references, and snapshots. We have seen how mutable variables in Cairo are equivalent to shadowed variables in Sierra and how references in Cairo use the ref

prefix to pass variables and implicitly return them. Additionally, we have seen how snapshots in Cairo are a unique concept that allows developers to keep ownership of objects while ensuring that the original value remains unmodified. Finally, we explored how developers can use function inlining as an optimization technique.

Understanding the core concepts of the Cairo stack is essential to becoming a better developer in the Starknet ecosystem, and we hope that this series has provided you with valuable insights and knowledge to improve your skills. Keep Starknet Strange and Flourishing!

About Us

Nethermind is a team of world-class builders and researchers. We empower enterprises and developers worldwide to access and build upon the decentralized web. Our work touches every part of the web3 ecosystem, from our Nethermind node to fundamental cryptography research and infrastructure for the Starknet ecosystem. Discover our suite of tools for Starknet:

[Warp, the Solidity to Cairo compiler

](https://nethermind.page.link/8a9f);

Voyager, a StarkNet block explorer; [Horus, the open-source formal verification tool

](https://nethermind.io/horus/)for Starknet smart contracts;

[Juno

](https://github.com/NethermindEth/juno), Starknet client implementation

, and

[Cairo Smart Contracts Security Auditing

](https://nethermind.page.link/25ee)services.

If you’re interested in solving some of blockchain’s most difficult problems, visit our

[job board

](https://nethermind.page.link/careers)!

Disclaimer

This article has been prepared for the general information and understanding of the readers, and it does not indicate Nethermind’s endorsement of any particular asset, project or team, nor guarantee its security. No representation or warranty, express or implied, is given by Nethermind as to the accuracy or completeness of the information or opinions contained in the above article. No third party should rely on this article in any way, including without limitation as financial, investment, tax, regulatory, legal or other advice, or interpret this article as any form of recommendation.